Padronização e Reuso de Aplicações Web com Demoiselle Framework

Uma plataforma de desenvolvimento do governo para toda a sociedade

Flávio Gomes da Silva Lisboa
(flavio.lisboa@serpro.gov.br): é bacharel em Ciência da Computação e cursa a especialização em Tecnologia Java na UTFPR. Trabalha como analista de Desenvolvimento na Coordenação Estratégica de Tecnologia do SERPRO, em Curitiba. Faz parte da equipe do projeto Demoiselle desde meados de 2008, documentando, codificando, elaborando testes e mais recentemente na construção da comunidade em torno do software. Também realiza consultoria interna e treinamentos na empresa.

A padronização do uso de softwares desenvolvidos para o Governo Federal abre a possibilidade de novas empresas de desenvolvimento de software, especialmente as de pequeno porte, trabalharem para o governo. Essa convergência digital oferece vantagens para a União, na medida em que permite que sistemas diferentes possam ser ligados com muita facilidade, e para o mercado, porque o governo dirá quais os padrões tecnológicos ele pretende usar e irá disponibilizar a ferramenta. Daremos um vislumbre da base dessa padronização, a plataforma Demoiselle para Java, construindo uma pequena aplicação web.

Demoiselle Framework é essencialmente uma biblioteca central de módulos que atende às necessidades de infraestrutura básica de uma aplicação web não distribuída. A ideia é que ele seja o mais fracamente acoplado quanto possível, de forma que através do desenvolvimento orientado a componentes, as aplicações possam ser criadas e modificadas com substituição ou acoplamento de novos módulos, sem que o núcleo central precise ser modificado.

Esse modelo de framework arquitetural permite fazer uma analogia com as montadoras de veículos. Os sistemas não precisam ser fabricados, desde o zero. Eles podem ser montados, a partir de uma infraestrutura genérica (o framework arquitetural) a qual são acoplados componentes que constituem a parte específica de cada sistema. O objetivo deste artigo é esclarecer como se dá a construção de uma aplicação web não distribuída em Java usando essa biblioteca central de módulos e um componente fornecido com o próprio framework (o qual não depende dele). Nosso estudo de caso será uma pequena agenda telefônica. Isso dará uma visão rápida e geral sobre o processo de desenvolvimento. Em seguida, destacaremos algumas características do Demoiselle usando exemplos do projeto Demoiselle Samples. O código-fonte completo do projeto da agenda pode ser baixado no site da revista Mundoj.

O nome

Demoiselle é o nome da melhor aeronave construída por Santos Dumont a qual influenciou significativamente a indústria da aviação, no começo do século XX. Com ele, Dumont realizou voos de até 18 km superiores aos 200m do 14 bis. O Demoiselle foi o primeiro avião fabricado em série no mundo, e se notabilizou por ser um projeto de "código livre", pois Santos Dumont permitia a livre utilização, adaptação e cópia de seu trabalho. Segundo Paul Hoffman, o idealismo de Dumont chegou a tal ponto que quando "aconselharam-no a patentear o Demoiselle", por dizer ele estar sem dinheiro, mesmo que ninguém acreditasse, "ele recusou. Era seu presente para a humanidade, disse que preferia terminar seus dias em um asilo de pobres que cobrar aos outros o privilégio de copiar sua invenção e de fazer experimentos aéreos". Para homenagear um dos brasileiros mais ilustres de todos os tempos, conhecido como Pai da Aviação, a plataforma de desenvolvimento de software baseada em software livre recebeu o nome de sua invenção, copiada, adaptada e que serviu de base para o desenvolvimento da aviação.

Objetivos do Demoiselle

Todos os sistemas comprados ou especialmente desenvolvidos para órgãos federais deverão utilizar a plataforma Demoiselle. As linguagens e outros aspectos que balizaram a elaboração da plataforma constituem padrões, o que facilitará sua adoção, não só por parte dos responsáveis pelas áreas de TI nos órgãos federais, como também pelas próprias empresas fornecedoras de soluções para o governo. Desse modo, o Governo Federal não precisará comprar produtos, mas sim contratar serviços para os quais não necessitará de treinamentos específicos.

A adoção do Demoiselle pretende automatizar e acelerar a integração de sistemas, aumentar a produtividade e eliminar o retrabalho. As premissas que nortearam a elaboração da Demoiselle estabeleciam que a plataforma deveria ser extensível, fácil de usar, estável, configurável, confiável e ter sua documentação publicada. A intenção era atingir padronização, redução da curva de aprendizagem, maior produtividade, simplificação dos processos, reutilização de códigos e uma manutenção mais simplificada.

Outro benefício esperado é a economia financeira, com o não-pagamento de licenças de software: a expectativa é de que haja redução de 50% nos custos de operação e manutenção dos sistemas. Outro benefício que pode ser apontado é que o governo não precisa se preocupar em fazer todo o desenvolvimento de seus sistemas, podendo contratar a iniciativa privada, controlando a tecnologia e ajustando a sua operação depois. Dessa forma, ganham tanto o governo quanto as empresas privadas.

Apesar de ter sido criado inicialmente para isso, o uso do Demoiselle Framework não se limita a aplicações para o governo, podendo ser utilizado livremente. Qualquer pessoa pode baixar o código do Demoiselle, criar aplicações e vendê-las, sem ter que investir em infraestrutura de software para desenvolvimento. A arquitetura de acoplamento fraco, orientada a componentes permite que qualquer desenvolvedor customize o Demoiselle para seus propósitos particulares. Ou seja, o Serpro está fomentando a indústria nacional de software ao oferecer para os desenvolvedores uma plataforma completa que integra diversas tecnologias e frameworks especialistas, além da experiência dos projetos em Java construídos no Serpro, presentes nos padrões adotados na codificação.

Essa oportunidade de negócio é assegurada pelo fato do Demoiselle ser disponibilizado sob a licença LGPL 3. Qualquer componente utilizado ou desenvolvido para ele deve ser compatível com essa licença. Mas a LGPL, ao contrário da GPL, não obriga os softwares gerados com o Demoiselle a serem livres e abertos. Essa flexibilidade dá ao desenvolvedor a liberdade de escolher se quer ou não abrir suas aplicações, permitindo a criação de softwares proprietários, conforme a conveniência ou necessidade.

Ressaltando o aspecto da licença, apesar de ter sido feito para atender ao Governo Federal, o Demoiselle é um software livre, e o seu desenvolvimento é feito de forma colaborativa. Assim você também pode ajudar no desenvolvimento e na evolução do Demoiselle, reportando erros, sugerindo melhorias, submetendo código, ajudando na documentação, traduzindo ou, simplesmente, dizendo que você está usando o framework!

Arquitetura

A estrutura geral do Demoiselle e suas dependências podem ser vistas na figura 1. Os componentes na verdade não fazem parte do framework arquitetural, pois possuem um ciclo de vida independente. Por isso mesmo não geram dependência obrigatória nas aplicações geradas, e podem ser construídos colaborativamente. A construção de componentes para o Demoiselle deve gerar diversos projetos vinculados a ele, desenvolvidos pela comunidade.

A arquitetura de referência proposta para o Demoiselle é baseada em camadas. Além das clássicas camadas do modelo MVC (Modelo, Visão e Controlador), elas se distinguem como camadas de persistência, transação, segurança, injeção de dependência e mensagem, essas últimas denominadas contextos, como veremos adiante.

As três camadas do Demoiselle são View & Controller, Business e Persistence.

Persistence corresponde à letra M do MVC, ou seja, aos modelos. É a camada onde residem as classes POJO, que podem ou não ter anotações de mapeamento, as classes DAO e os filtros de pesquisa.

View & Controller é a camada literalmente correspondente às letras V e C do MVC. Nessa camada residem as classes responsáveis pela interface gráfica da aplicação e pelo controle dos eventos.

O Demoiselle cria mais uma camada chamada Business, onde reside a lógica de negócios da aplicação.

Módulos

- O módulo Core contém o conjunto de especificações que dão base estrutural ao framework possibilitando padronização, extensão e integração entre as camadas das aplicações nele baseadas.
- O módulo Persistence realiza a integração do sistema com outros sistemas gerenciadores de dados garantindo eficiência para recuperar, armazenar e tratar informações.
- O módulo útil contém componentes utilitários que facilitam o trabalho de outras funcionalidades do framework e seus módulos lógicos.
- O módulo View contém implementações de componentes específicos de interface com usuário baseados na especificação JSF.
- O módulo Web é a implementação do módulo lógico Core para aplicações Web (J2EE), prover utilitários comuns de aplicações web que facilitam tratamento de sessões de usuário e suas requisições.

Figura 1. Arquitetura do Demoiselle e dependências.

Contextos

A criação de camadas visa diminuir a complexidade da aplicação, separando-a em partes mais compreensíveis, substituíveis e fáceis de manter. Geralmente usamos metáforas com bolos e tortas, onde uma camada repousa sobre a outra. Nessa metáfora, as camadas de software só podem se comunicar com as camadas adjacentes, imediatamente "acima" ou "abaixo". O Demoiselle separa as aplicações em camadas, representadas na implementação por pacotes Java, seguindo esse modelo.

Cada camada é responsável por um trabalho específico, tornando-se especialista em um conjunto de tarefas (ou serviços). Há, no entanto, algumas operações que permeiam todas as camadas, como as transações e as mensagens da aplicação. Há também o controle da segurança da aplicação, que deve abarcar todas as camadas. E, finalmente, existe a necessidade de tornar as camadas as mais independentes possíveis umas das outras, de modo a permitir que cada uma constitua um componente reutilizável. Para tal, é necessário que não haja dependência explícita no código, o que pode ser feito mediante o uso de programação orientada a aspectos.

A orientação a aspectos, além de conseguir evitar as repetições de código intratáveis pela orientação a objetos, centralizando blocos de código e facilitando a manutenção, permite que os trechos de código que geram dependência entre camadas sejam injetados somente em tempo de compilação. No Demoiselle Framework, a orientação a aspectos é realizada pelo AspectJ, uma extensão para a linguagem Java que inclui uma linguagem orientada a aspectos e um compilador.

Toda essa infraestrutura para as camadas tradicionais constitui o que chamamos de contextos. Em uma representação gráfica (figura 2) os contextos podem ser vistos como camadas horizontais, enquanto a implementação de MVC do Demoiselle constitui um conjunto de camadas transversais.
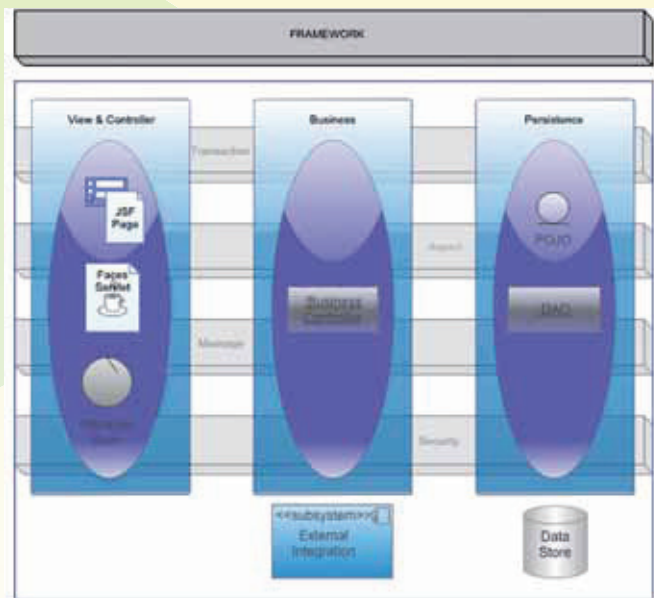
Figura 2. Camadas verticais e transversais do Demoiselle.

Integração entre camadas

Entre os aspectos a destacar sobre a arquitetura, o mais significativo é a utilização de padrões de projeto, tais como Factory, Proxy, IoC e injeção de dependências para manter a integração de camadas em um nível de acoplamento baixo a fim de garantir uma melhor manutenção e escrita/legibilidade das classes representantes de cada camada.

O mecanismo de integração entre camadas atua na camada de visão injetando objetos de negócio através de uma fábrica do próprio framework ou alguma fábrica definida pela aplicação e esta fábrica poderá utilizar um proxy, do framework ou da aplicação, para a instanciação do objeto de negócio.

O mecanismo de integração entre camadas atua também na camada de regras de negócio injetando objetos de persistência através de uma fábrica do próprio framework ou alguma fábrica definida pela aplicação e esta fábrica poderá utilizar um proxy, do framework ou da aplicação, para a instanciação do objeto de persistência.

Oportunidades de Negócio

Figura 3. Benefícios e oportunidades do Demoiselle.

A princípio, o Demoiselle pode ser visto como uma imposição, com o intuito de criar uma reserva de mercado para empresas estatais. Mas isso é um ledo engano, pois como o software é aberto, qualquer um pode estudá-lo, melhorá-lo e oferecer serviços relacionados a ele, seja desenvolvimento, manutenção, treinamento ou consultoria.

À primeira vista parece ser mais um framework Java, o que pode levantar questões sobre o motivo de ter sido criado, em vez de se adotar um já existente como padrão. A intenção não é criar um produto para competir com outros frameworks, mas estabelecer uma plataforma que implemente o conceito de framework integrador, que realiza a integração entre vários frameworks especialistas e garante a evolução, manutenibilidade e compatibilidade entre cada um deles. Como visto na figura 1, o Demoiselle não reinventa a roda, mas tão somente faz vários frameworks especialistas, consagrados pelo mercado, trabalharem em uníssono. Sua maior contribuição é dar um direcionamento ao uso das tecnologias.

O direcionamento tecnológico é importante para os prestadores de serviço, pois permite que eles se especializem nas tecnologias que foram definidas como o padrão a ser utilizado. Com a adoção de uma arquitetura de referência e uma plataforma integradora de tecnologias, um órgão do Governo Federal pode contratar uma empresa para desenvolver um sistema e depois ter a segurança de contratar outra para dar manutenção. Ou ainda, se tiver uma área de desenvolvimento de sistemas, pode ele mesmo realizar manutenção. Isso permite contratações mais flexíveis, assim como licitações de ampla concorrência.

Dessa forma, pequenas e médias empresas, ou até desenvolvedores ou consultores independentes que trabalhem como pessoa jurídica, podem participar de concorrências públicas junto com grandes empresas, pois todos terão acesso ao Demoiselle e às tecnologias relacionadas sem qualquer custo. O Governo Federal ganha, pois a ampla concorrência tende a diminuir os custos de manutenção dos sistemas de informação, além do que a própria plataforma é gratuita, o que o desonera em relação a licenças. A iniciativa privada ganha como um todo, pois a padronização dos sistemas democratiza a concorrência, evitando que um grupo de privilegiados tenha o mercado governo reservado para si. Como diria o matemático John Nash, "o melhor resultado virá quando todos do grupo fizerem o melhor para si mesmos e também para o grupo como um todo".

Mas o Demoiselle extrapola as relações comerciais entre governo e iniciativa privada. Qualquer empresa pode utilizar a plataforma para desenvolver sistemas para qualquer cliente. E esse cliente terá a segurança de que não dependerá do criador da aplicação para mantê-lo. Não ficará refém ou prisioneiro, mas terá liberdade de tomar conta de seus sistemas de informação e de implementar funcionalidades sem ter que esperar por próximas versões. Tudo o que alguém precisará para fazer manutenção em uma aplicação desenvolvida pelo Demoiselle será conhecer os requisitos, o framework e a arquitetura de referência, sendo que somente os primeiros irão mudar a cada caso.

E apesar de inicialmente atender ao desenvolvimento de aplicações web não distribuídas, o Demoiselle evoluirá de modo a servir aplicações distribuídas, desktop e acessíveis e embarcadas em dispositivos móveis, ampliando o leque de oportunidades de negócio e criando uma referência para o uso de tecnologias baseadas em Java.

A qualidade é claramente afetada pelo estabelecimento de uma comunidade de usuários da plataforma, pois efetiva o reuso de componentes de software em vez da recriação dos mesmos. O Demoiselle oferece uma oportunidade de agregar a experiência de diversas comunidades de software relativas a cada um dos frameworks especialistas e criar uma grande rede de colaborativa de desenvolvimento que consiga convergir seus objetivos e necessidades individuais de modo que todos possam se beneficiar.

Ambiente de desenvolvimento

O ambiente compatível para o desenvolvimento baseado no Demoiselle é composto de JVM, IDE e servidor de aplicação. A JDK requerida é a 1.5.x. Como IDE, recomendamos o Eclipse 3.3.2 JEE Developer (Europa) com o plugin AJDT 1.5.2.200804241330. Neste artigo, usaremos o Eclipse. Os servidores JEE com contêiner web 2.5 são compatíveis com o Demoiselle. Isso inclui Tomcat 6.x e JBoss 4.2.x.

Iniciando

Configure o Eclipse para trabalhar com o servidor escolhido e baixe os pacotes do framework. Tanto eles quanto os demais artefatos que forem necessários poderão ser baixados do site da Mundoj. Crie um projeto chamado addressbook, no pacote br.gov.demoiselle.samples, seguindo a estrutura das figuras 4 e 5. Essa estrutura não é totalmente obrigatória na verdade, mas é a referência para implementação com o Demoiselle. Não mostraremos todo o código da aplicação neste artigo, mas ele estará disponível para download no site da Mundoj.
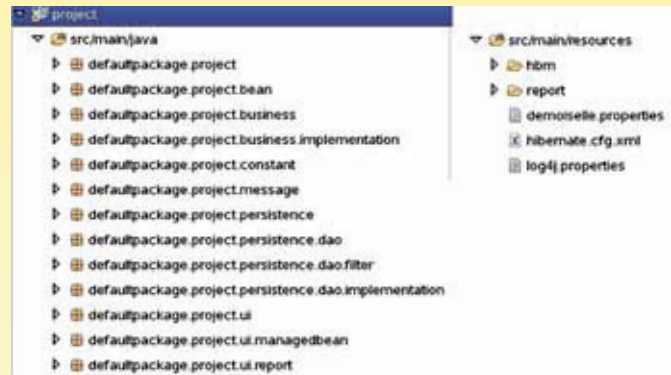
Figura 4. Estrutura da pasta main.
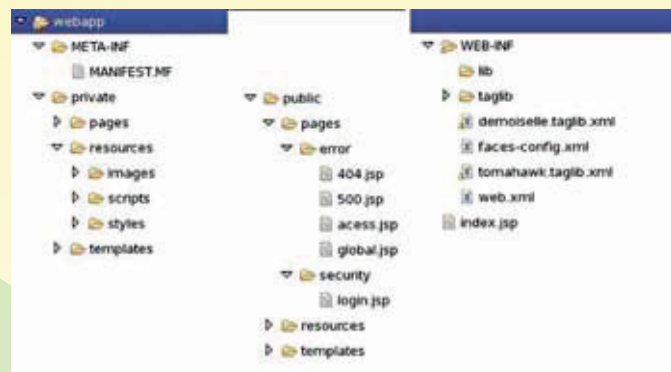
Figura 5. Estrutura da pasta webapp.

POJO

Serão criadas duas classes POJO (figura 7) referentes às tabelas do banco de dados agenda (figura 6).
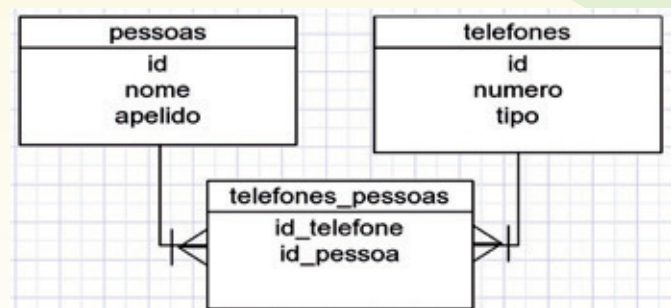
Figura 6. DER do banco agenda.
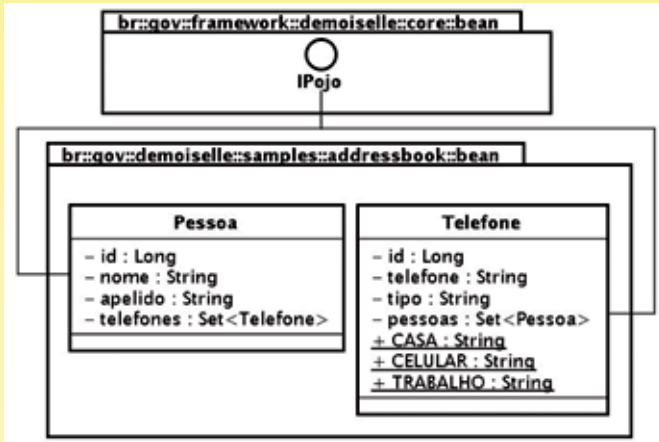
Figura 7. Classes POJO do projeto addressbook.

Camada de persistência

O módulo Persistence do Demoiselle disponibiliza abstrações de persistência visando o baixo acoplamento nas camadas superiores e a fácil utilização de frameworks de persistência, como Hibernate e JDBC. Nesse exemplo, será utilizado o Hibernate.

Para implementar a persistência, construa o mapeamento objeto-relacional da aplicação, configure a conexão, referencie o mapeamento e finalmente crie as interfaces para a classes DAO. Isso é necessário porque as classes DAO não serão instanciadas diretamente nas camadas superiores, pois será feito o uso de injeção de código por meio do AspectJ.
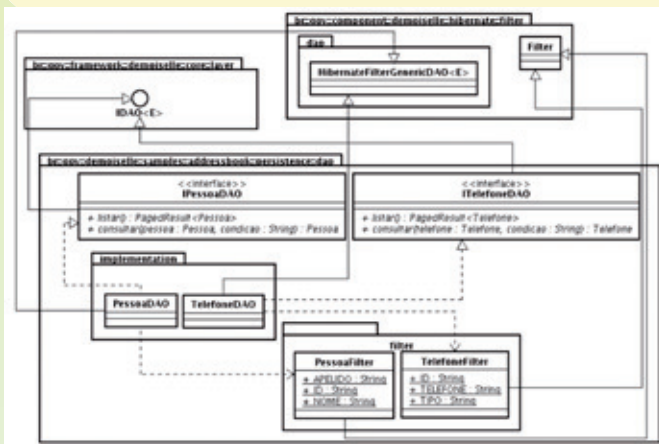
Figura 8. Camada de persistência.

Conforme pode ser visto na figura 8, a implementação das interfaces será feita por uma classe que estende HibernateFilterGenericDAO. Esta é uma classe que já traz encapsulada a lógica de persistência utilizando o framework Hibernate, além de agregar a funcionalidade de filtros para consultas. Essa classe não faz parte do framework arquitetural, e sim de um componente desacoplado, o Demoiselle-Hibernate-Filter. O Demoiselle traz uma classe chamada HibernateGenericDAO, mas esta não possui filtros implementados.

HibernateFilterGenericDAO é a extensão de HibernateGenericDAO. Esse é um exemplo claro da arquitetura orientada a componentes. O framework arquitetural traz construções o mais genéricas possíveis, para evitar o engessamento das aplicações. As alterações e extensões de funcionalidades devem ser feitas pelos componentes.

A Listagem 1 traz a implementação dos métodos listar e consultar da classe PessoaDAO.

Listagem 1. Métodos de PessoaDAO.

```
public PagedResult<Pessoa> listar(Page page) {
  PessoaFilter f = new PessoaFilter();

  PagedResult<Pessoa> results = find(f, page);

  if (results != null && results.getTotalResults() > 0) {
    return results;
  }
  return null;
}

public Pessoa consultar(Pessoa pessoa,String filtro) {
  PessoaFilter f = new PessoaFilter();
  Object value;

  filtro = filtro == null ? PessoaFilter.ID : filtro;
  value = filtro.equals(PessoaFilter.ID) ? pessoa.getId() :
    (filtro.equals(PessoaFilter.NOME) ? pessoa.getNome() :
    pessoa.getApelido());

  f.addEquals(filtro, value);

  List<Pessoa> results = find(f);

  if (results != null && results.size() > 0) {
    return results.get(0);
  }
  return null;
}
```

Perceba que o método listar dessa classe faz uso de uma instância de PessoaFilter (Listagem 2). Essa é uma especialização da classe Filter disponibilizada pelo componente Demoiselle-Hibernate-Filter. Ela abstrai a construção de consultas em SQL. No construtor das filhas de Filter, deve ser invocado o método setClazz(), passando como parâmetro o atributo class do POJO relacionado.

O método find(), usado por pessoaDAO, é herdado de HibernateFilterGenericDAO. Ele encapsula as consultas SQL para a base de dados. No método listar(), são passados dois argumentos, um objeto Filter e um objeto Page. O primeiro objeto é um filtro vazio, o que irá provocar o retorno de todos os dados da tabela relacionada. O segundo é uma classe implementada pelo módulo útil do Demoiselle, que permite a paginação de resultados de consultas. A paginação é uma técnica muito útil (sem trocadilhos), pois permite dividir o resultado de uma consulta em segmentos denominados páginas, e recuperar somente a que interessa naquele momento. Então, embora a consulta traga todos os resultados, pelo fato do filtro não especificar nenhuma condição, somente a página solicitada será devolvida.

No método consultar(), temos outra assinatura de find(). Neste caso, o filtro recebe uma condição, que é um teste de igualdade, e somente ele é passado como argumento para find(). Nessa assinatura recebemos de volta uma coleção List da classe POJO Pessoa.

Listagem 2. Classe PessoaFilter.

```
package br.gov.demoiselle.samples.addressbook.persistence.dao.filter;

import br.gov.component.demoiselle.hibernate.filter.Filter;
import br.gov.demoiselle.samples.addressbook.bean.Pessoa;

@SuppressWarnings("serial")
public class PessoaFilter extends Filter{
  public static final String ID = "id";
  public static final String NOME = "nome";
  public static final String APELIDO = "apelido";

  public PessoaFilter() {
    setClazz(Pessoa.class);
  }
}
```

Camada de negócio

Esta camada contém a implementação da lógica de negócios da aplicação. É aqui que se concentra o maior esforço do desenvolvedor. Seguindo com nosso exemplo, criaremos a classe ContatoBC, a qual fará uso das classes de persistência da seção anterior. Como o Demoiselle trabalha com injeção de dependências, nós sempre programamos para interfaces. Assim, criamos inicialmente a interface IContatoBC e em seguida a implementamos na classe ContatoBC (figura 9).
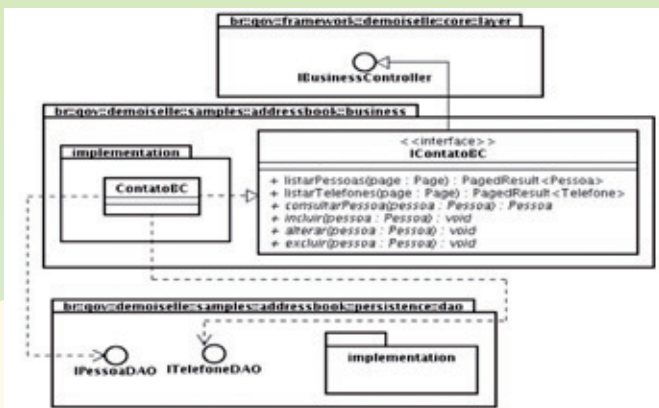
Figura 9. Camada de Negócios sem dependência da implementação da Camada de Persistência.

Listagem 3. Injeção de código em ContatoBC.

```
@Injection


IContatoDAO contatoDAO;
```

Conforme vemos na Listagem 3, não criamos instância da classe ContatoBC. O próprio Demoiselle, por meio de um aspecto associado à interface IBusinessController, a qual é estendida por IContatoBC, injeta o código de instanciação em tempo de compilação. Por isso, a anotação @Injection antes do atributo contatoDAO. Isso permite que a classe implementadora da interface seja alterada, removendo a dependência dos BCs das classes DAO. A localização da implementação é baseada no nome da interface, por isso as convenções de nome. Mas o Demoiselle permite que ao desenvolvedor definir suas próprias fábricas de objeto, alterando esse comportamento padrão.

Para ressaltar o quanto a injeção de código por aspectos ajuda na independência entre camadas, vamos examinar o método incluir() de ContatoBC (Listagem 4).

Listagem 4. Método incluir de ContatoBC.

```
public void incluir(Pessoa pessoa) {
  pessoaDAO.insert(pessoa);
}
```

É claro que essa implementação é o exemplo mais reduzido, pois antes de chamar o método insert() da classe DAO, uma série de operações poderia ser feita, como validação e filtragem de dados, por exemplo. Mas o que importa aqui é ver que chamamos o método insert() de um objeto cuja classe só será definida quando o AspectJ compilar o código. Desde que a classe do objeto implemente a interface IPessoaDAO, não importa para o controlador de negócios como é realizado o acesso aos dados na camada de persistência.

Neste caso, o método insert() chamado será o da classe PessoaDAO (Listagem 5). Vemos que esse método faz uso da classe HibernateUtil, que como sugere o nome, utiliza o Hibernate. Mas e se nossa aplicação tiver de usar outro framework de ORM ou mesmo fizer uso direto de JDBC? Ou se quisermos reaproveitar a camada de negócio em outra aplicação, com outra implementação de persistência? Não há problema, porque o controlador de negócio ignora isso completamente. Para ele o que importa são as interfaces, não as implementações.

Listagem 5. Método incluir de ContatoBC.

```
public Object insert(Pessoa arg0) {

  Object object = super.insert(arg0);

  HibernateUtil.getInstance().getSession().flush();

  return object;

}
```

Camada de apresentação

Em uma analogia com a Idade Média Ocidental, esta camada representa o exterior de um castelo, cercado por um fosso com água e altas muralhas com arqueiros posicionados, onde o único meio de acesso é a ponte levadiça. O controle de navegação de páginas (ou telas) é realizado por esta camada, bem como o controle sobre a entrada e saída de dados da aplicação.

ManagedBean

O ator principal dessa camada é o ManagedBean. Em nosso exemplo, criaremos a classe ContatoMB (figura 10). A instância do BusinessController é injetada pelo aspecto associado à classe AbstractManagedBean (Listagem 6).

Listagem 6. Injeção de código em ContatoMB.

```
@Injection
IContatoBC contatoBC;
```
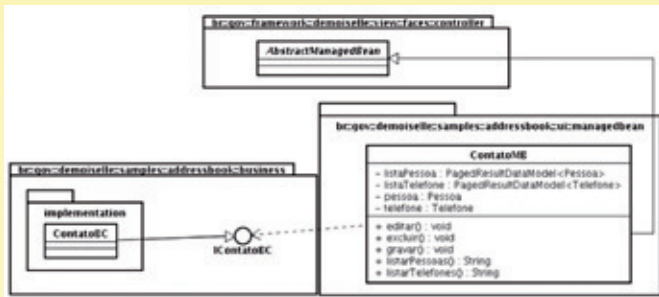
Figura 10. Camada de Apresentação sem dependência da implementação da Camada de Negócio.

Agora é só escolher seu servidor web favorito, desde que siga a especificação Servlets 2.5, e adicionar seu projeto a ele. JBoss 4.2.x e Tomcat 6.x são compatíveis. Antes de iniciar (ou reiniciar o servidor), lembre-se de incluir alguns registros na tabela, pois para deixar este exemplo o mais curto possível, não tratamos essa exceção.

Com o código que construímos, você pode criar uma página de contatos telefônicos, tal como a da figura 11.

Figura 11. Página de contatos telefônicos.

Detalhamento da arquitetura

O exemplo que vimos aqui é algo extremamente simples, porém seu desenvolvimento sem o uso de um framework tomaria mais tempo. A questão agora é por que usar o Demoiselle quando já existem outros frameworks no mercado? Quais são seus diferenciais? Veremos agora mais detalhes sobre o módulo Core, o centro nervoso do Demoiselle e o módulo Web, que faz a implementação para aplicações Web.

Os exemplos de código mostrados a seguir fazem parte da aplicação escola, um exemplo disponível no projeto Demoiselle Samples (http://sourceforge.net/projects/demoiselle-samp).

Injeção de dependências

Como dito anteriormente, o módulo Core contém as especificações que dão base estrutural ao framework. Ele define quatro interfaces:

- IViewController – abstração para o objeto da camada de visão;
- IBusinessController – abstração para o objeto da camada de negócio;
- IDAO – abstração para o objeto da camada de persistência;
- IFacade – abstração para o objeto da camada de integração de módulos/subsistemas.

A lógica de integração entre camadas reside no pacote br.gov.framework.demoiselle.core.layer.integration. Ele utiliza padrões de projeto, tais como: Factory, Proxy, IoC e injeção de dependências para manter a integração de camadas com baixo nível de acoplamento, a fim de garantir melhor manutenção, escrita e legibilidade das classes representantes destas camadas.

No exemplo da agenda, nós vimos a forma mais simples de injeção, mas a anotação Injection pode receber argumentos, o que torna flexível a instanciação de objetos, como vemos nas Listagens 7 e 8.

Listagem 7. Exemplos de Injeção de um IBusinessController.

```
public  class MeuMB implements IViewController{
@Injection
private IMeuBC meuBC;
}

public  class MeuMB implements IViewController{
@Injection (name="br.gov.escola.business.implementation.AlunoBC")
private IMeuBC meuBC;
}
```

Listagem 8. Exemplos de Injeção de um IDAO.

```
public  class MeuBC implements IBusinessController{
@Injection
private IMeuDAO meuDAO;
}

public  class MeuBC implements IBusinessController{
@Injection(
name="br.gov.escola.persistence.dao.implementation.AlunoDAO")
private IMeuDAO meuDAO;
}
```

O que é importante compreender é que o módulo Core especifica quem trata a injeção de dependência. Os módulos que implementam o Core é que devem definir como a injeção será realizada.

Na versão 1.0.x do Demoiselle a injeção de dependência é implementada no módulo Web, usando AOP para detectar os pontos de integração, como já dissemos. O que significa que outros tipos de aplicação que não forem Web não distribuídas deverão implementar a injeção de dependência em seus respectivos módulos. Isso é algo que se espera que ocorra com o desenvolvimento colaborativo em comunidade.

Contexto de mensagens

O Demoiselle define uma abstração de mensagens trocadas durante uma requisição entre as camadas do sistema. Isso é realizado pelo seguinte trio:

- Interfaces
  - IMessage – abstração da unidade de mensagem.
  - IMessageContext – abstração do contexto de mensagem.
- Enumerations
  - Severity – lista de severidades.

O módulo Web implementa o contexto de mensagens com a classe WebMessageContext. Na Listagem 9 temos um exemplo de adição e recuperação de mensagens no contexto de segurança, que será tratado a adiante.

Listagem 9. Manipulação de mensagens no Demoiselle.

```
ISecurityContext contextoMsg = ContextLocator.getInstance().
getSecurityContext();

public class MeuBC implements IBusinessController {
  public void meu_metodo(){
    ...
       contextoMsg.addMessage(InfoMessage.Mensagem);
  }

  public class MeuMB extends IViewController {
    public void meu_metodo(){
      for  (IMessage imsg : contextoMsg.getMessages(){
        addMessage(imsg);
      }
```

Tratamento de exceção

O módulo Core define uma exceção padrão para as aplicações, ApplicationRuntimeException. Esta exceção, do tipo "unchecked", encapsula uma mensagem padronizada para facilitar o tratamento pelos módulos do aplicação (Listagem 10).

Listagem 10. Exemplos de tratamento de exceção com Demoiselle.

```
public void MetodoBC(){
  if ( /*Condição para lançamento de exceção*/ ){
    throw new ApplicationRuntimeException(ErrorMessage.ERRO_01);
  }
}
public void MetodoMB() {
  try {
    MetodoBC();
  } catch (ApplicationRuntimeException ex) {
    /**Trata exceção*/
  }
}
```

Contexto de segurança

O Demoiselle especifica um mecanismo padrão para acesso a dados de segurança referentes à autenticação e autorização. A autorização é feita por meio de papéis (Listagem 11). Ele faz isso por meio do conjunto de APIs JAAS que permitem às aplicações escritas na plataforma J2EE usar serviços de controle de autenticação e autorização sem necessidade de ficarem dependentes dos mesmos.

O módulo Web implementa o contexto de segurança proposto no módulo Core através de um Singleton. O contexto de segurança é inicializado a cada requisição do usuário com informações de autenticação e autorização.

Listagem 11. Exemplo de uso do Contexto de Segurança.

```
ISecurityContext contexto = ContextLocator.getInstance().

getSecurityContext();


if  (contexto.isUserInRole("Administrador")){

...

}
```
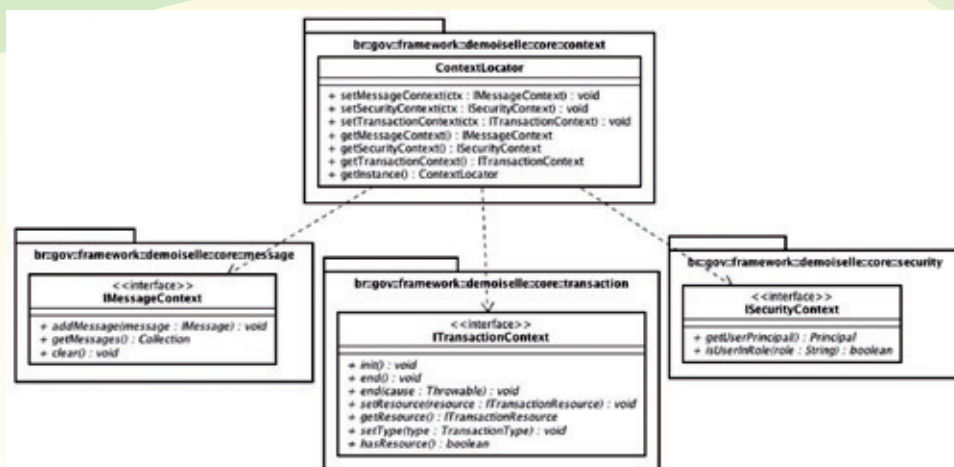
Figura 12. Localizador de contextos.

Entidades

O Core do Demoiselle propõe uma abstração para as entidades da aplicação, a interface IPojo. Como tudo no Core, é algo muito genérico, feito com o intuito de estabelecer padrões. Mas para os que querem algo mais elaborado, o componente Demoiselle-Commons traz uma abstração POJO baseada em uma extensão de IPojo, PojoExtension. Outros componentes que não são obrigatórios, mas podem ser usados com o framework, estão disponíveis em http://sourceforge.net/projects/demoiselle-comp, o projeto para componentes do Demoiselle Framework. Você já está convidado a contribuir com o seu.

Controle de transação

O Core especifica o mecanismo de controle transacional e define um contexto transacional que atua no início e no fim de cada ação. Seu funcionamento depende de um tipo definido, seja Local ou JTA.

- Local – indica que a aplicação será responsável pelo gerenciamento da transação.
- Distribuída (JTA) – a aplicação dependerá de uma implementação JTA disponível no contêiner.

Acionadores

O Core define um mecanismo padronizado de ações a serem executadas pela aplicação. Essas ações são definidas como funções estruturais da aplicação, a saber:

- carregamento de configuração;
- inicialização de ambiente;
- etc.

Localizador de contextos

Para que a aplicação possa usufruir dos contextos definidos no módulo Core, a existência de um localizador é fundamental. A implementação de cada contexto (segurança, transação, mensagem etc.) deverá utilizar o localizador como canal de acesso. A estrutura do localizador e seu relacionamento com os contextos podem ser vistos no diagrama de classes da figura 12.

Redirecionamento

O módulo Web implementa um mecanismo de redirecionamento baseado em URL, que é utilizado por componentes do Demoiselle como o Report, mas pode também ser utilizado pelas aplicações. As Listagens 12, 13, 14 e 15 mostram os passos para criar um redirecionamento.

Listagem 12. Criando uma IRedirectAction.

```
public class MinhaRedirectAction implements IRedirectAction {

  private ServletRequest request;
  private ServletResponse response;

  public String getParameter() {
    return "MinhaActionParameter";
  }

  public String getValue() {
    return "MinhaActionValue";
  }

  public void setRequest(ServletRequest req) { this.request = req; }

  public void setResponse(ServletResponse resp) { this.response = resp; }

  public void execute() {
    /*Minha execução*/
  }
}
```

Listagem 13. Cadastrando a Ação de Redirecionamento no demoiselle.properties.

```
# --- Web Configuration ---

framework.demoiselle.web.redirect.action=MinhaRedirectAction01

framework.demoiselle.web.redirect.action=MinhaRedirectAction02

framework.demoiselle.web.redirect.action=MinhaRedirectAction03
```

Listagem 14. Configuração do web.xml.

```
<servlet>
  <servlet-name>WebRedirectServlet</servlet-name>
  <servlet-class>
    br.gov.framework.demoiselle.web.redirect.WebRedirectServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>WebRedirectServlet</servlet-name>
  <url-pattern>/redirect</url-pattern>
</servlet-mapping>
```

Listagem 15. Chamando a RedirectAction.

```
<a href="minhaAplicacao/redirect?MinhaActionParameter=MinhaActionValue">

  Chamar Minha Action

</a>
```

Muito mais que isso

A ocorrência de mudanças deve ser a única certeza que o desenvolvedor de sistemas tem sobre o futuro. Por isso, a preocupação do Demoiselle em estar baseado em uma arquitetura de fraco acoplamento. A arquitetura de fraco acoplamento representa uma solução de flexibilidade similar ao que o padrão Decorator é para o problema de herança na orientação a objetos. Assim como o Decorator consegue prover poderes a um objeto escapando da dependência e insuficiência da herança, a arquitetura de fraco acoplamento consegue prover uma infraestrutura básica de funcionalidades com o uso de componentes.

Essa é "a deixa" para falar que o Demoiselle é mais do que apenas o projeto do framework arquitetural. Hoje o portal do Demoiselle Framework elenca quatro subprojetos:

- **Demoiselle Wizard:** plugin para o Eclipse que disponibiliza um menu com opções para gerar código automaticamente;
- **Demoiselle Sample:** projeto de uma aplicação-exemplo chamada escola;
- **Demoiselle Component:** pacote para desenvolvimento de componentes a serem acoplados ao framework. Já dispõe de vários componentes desenvolvidos e usados pelo Serpro;
- **Demoiselle Process:** a ideia desse projeto é sugerir um processo para construção de aplicações usando o Demoiselle.

Considerações finais

Vimos o contexto de criação do Demoiselle Framework, seus objetivos, sua proposta, sua característica de software livre de desenvolvimento colaborativo e finalmente criamos uma pequena aplicação web com o mesmo. Obviamente uma aplicação muito simples, mas completa no tocante à utilização das camadas propostas pelo framework. Ela deve ser suficiente para dar uma dimensão do que o Demoiselle oferece e instigar os leitores a completá-la, melhorá-la, levando-os, é claro, a estudar a documentação do mesmo.

O Demoiselle apenas está em consonância com o ideal de um mundo de padrões. Isso não tem a ver com uma visão de Admirável Mundo Novo, de Aldous Huxley (e muito menos de 1984 de George Orwell). Padrões são bons, quando bem utilizados. Eles permitem que as pessoas se comuniquem. Permitem que peças sejam fabricadas por pessoas diferentes e, mesmo assim, possam ser integradas em um único produto final. Assim como o treinamento constante faz o lutador marcial melhorar seu desempenho a cada dia, o desenvolvimento de um software com a mesma plataforma e processo tende a provê-lo de melhorias pela experiência acumulada com o tempo.

O melhor exemplo do que a falta de padrões pode ocasionar é o nosso sistema notarial. Quando você precisa do serviço de um cartório, pode embarcar numa grande aventura. Quando precisa do serviço de vários, isso pode se transformar em uma verdadeira saga. Uma saga em um sistema onde cada cartório parece ser um país diferente, com leis diferentes, no qual não há cooperação, você não tem certeza das regras e o mais importante: você não sabe exatamente quanto vai gastar até ter concluído o processo. Como diria a menina de lindos olhos claros da série Head First: 'Como seria maravilhoso se houvesse uma plataforma de desenvolvimento que fosse mais atrativa do que ter que reconhecer firma em cinco cartórios diferentes... deve ser um sonho'. Talvez seja um sonho. Ou talvez seja o Demoiselle• MJ

Saber mais

Mundoj Edição 21. Mundo OO: Design Patterns para um Mundo Real – parte 1.

Mundoj Edição 22. Mundo OO: Design Patterns para um Mundo Real – parte 2.

Mundoj Edição 23. Mundo OO: Design Patterns para um Mundo Real – parte 3.

Mundoj Edição 22. Padrões de Projeto e Reflexão. Alexandre Gazola.

Mundoj Edição 21. Conhecendo JSF e Facelets 1.2.

Mundoj Edição 19. Reflexão + Anotações – Uma Combinação Explosiva.

Mundoj Edição 18. O reuso na prática.

Referências

- *Sítio oficial do Demoiselle Framework – http://www.frameworkdemoiselle.gov.br*
- *Página do projeto no Sourceforge – http://sourceforge.net/projects/demoiselle*
- *Fowler, Martin. Padrões de Arquitetura de Aplicações Corporativas. Porto Alegre. Bookman, 2006.*
- *Hoffman, Paul. Asas da Loucura: a Extraordinária Vida de Santos-Dumont. Rio de Janeiro. Objetiva, 2004.*
- *Macias, Ananda. Framework de Desenvolvimento – Visão Geral. Disponível em <http://www.serpro.gov.br/clientes/serpro/serpro/imprensa/publicacoes/tematec/2008/ttec92_a>*
- *Winck, Diogo V. e Goeten Junior, Vicente. AspectJ – Programação Orientada a Aspectos com Java. São Paulo. Novatec, 2006.*
- *Howard, Ron. Uma Mente Brilhante. Baseado no livro homônimo de Sylvia Nasar, ganhador de quatro Oscar, incluindo Melhor Filme. 2001.*